Universidade da Beira Interior

Programação Funcional

Simão Melo de Sousa

Aula 4 - Introdução à programação funcional

- breakout game
- tartaruga Logo
- tocar uma partitura musical
- árvores quaternárias
- o problema das oito rainhas
- conclusão. Quer saber mais?

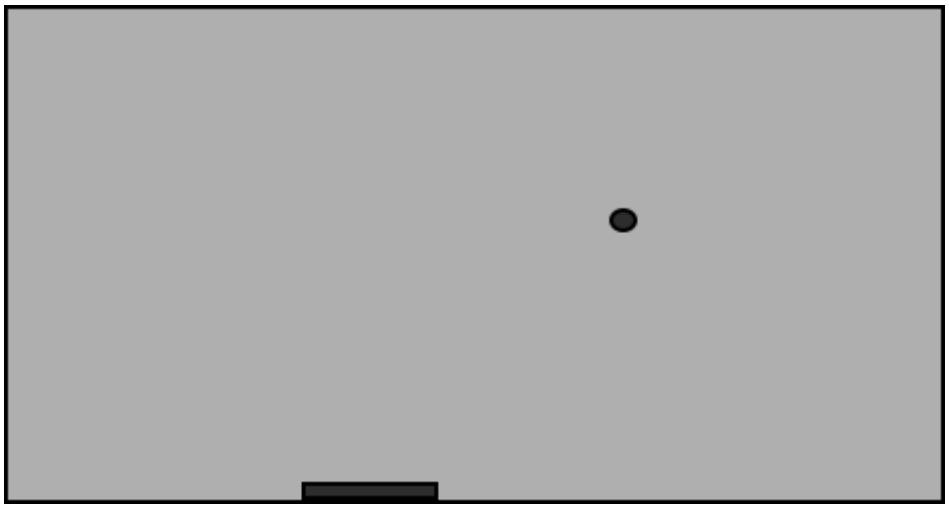
Breakout game

noções por introduzir neste exemplo

- unidades de compilação
- compilação separada
- módulos e interface

o objectivo é criar um jogo muito simples onde uma bola evoluí numa zona rectângular com três paredes

a bola pode cair na zona inferior da zona se não for impedida por uma raquete que ali desliza horizontalmente sob comando do jogador (aqui, via acção do rato)



o jogo repousa sobre alguns ingredientes de base que convém atualizar e controlar

1. é preciso definir três constantes: a dimensão do tabuleiro (i.e. os píxeis, decorrentes da altura e largura), o tamanho da bola (i.e. o seu raio) e da raquete (i.e. altura, largura)
2. estado do jogo: dado um momento, onde estão raquete e bola? qual o vector de velocidade da bola?
3. o algoritmo de atualização de posição/jogo que descrevemos já a seguir: como muda o estado do jogo de um momento para o momento seguinte? como tomar conta da ação do jogador?

usamos as facilidades do módulo `Graphics`, em particular a posição $(0, 0)$ está no canto inferior esquerdo

algoritmo do jogo

1. inicializar o estado do jogo (posição e vector de velocidade da bola, posição da raquete)
2. apagar a janela gráfica
3. calcular a posição da raquete em função do rato
4. mostrar a bola e a raquete
5. calcular o novo estado do jogo, isto é:
   - a nova posição da bola, em função das suas actuais posição e vector de velocidade
   - o novo vector de velocidade da bola, en função dos eventuais ressaltos nas paredes ou então na raquete
6. voltar ao ponto 2.

este algoritmo tem ações que se classificam em duas categorias

- as operações gráficas
- as operações de cálculo sobre o estado do jogo

em consequência, faz sentido repartir o código em dois suportes distintos: os ficheiros `draw.ml` e `breakout.ml`: são **unidades de compilação**
três de várias justificações para tal são:

1. de uma forma geral, é boa prática separar o código em vários módulos/ficheiros logicamente coherentes (i.e. em que cada módulo tem uma função bem definida)
2. separar a visualização do cálculo permite, se necessário e por exemplo, mudar a visualização de forma transparente sem nenhum impacto no ficheiro de cálculo
3. podemos imaginar que a visualização possa ser usada noutros contextos que esse aqui: possibilitamos reutilização

breakout game - draw.ml

```
open Graphics
let left = 0.
let right = 300.
let down = 0.
let up = 200.

let ball = 5
let paddle = 50
let thick = 8

let gray = rgb 220 220 220

let init () =
  let s = Printf.sprintf
            " %dx%d"
            (truncate right)
            (truncate up)    in
  open_graph s;
  auto_synchronize false
```

```
let clear () =
  set_color gray;
  fill_rect 0 0 (truncate right)
                (truncate up)

let get_paddle_pos () =
  let x = fst (mouse_pos ()) in
  max 0 (min x (truncate right - paddle))

let game x y =
  clear ();
  set_color black;
  fill_circle (truncate x)
              (truncate y) ball;
  let x = get_paddle_pos () in
  fill_rect x 0 paddle thick;
  synchronize ();
  x
```

breakout game - breakout.ml

```
let bounce (x, y) (vx, vy) xp =
  let vx =
    if x <= Draw.left || x >= Draw.right
    then -. vx else vx in
  let vy =
    if y <= float Draw.thick && x >= xp
       && x <= xp +. float Draw.paddle
       || y >= Draw.up
    then -. vy else vy
  in
  (vx, vy)

let new_position (x, y) (vx, vy) =
   x +. vx, y +. vy
```

```
let rec play (x, y) (vx, vy) =
  if y <= Draw.down
    then failwith "game over";
  let xp = Draw.game x y in
  let vx, vy = bounce (x, y) (vx, vy)
                      (float xp) in
  let x', y' = new_position (x, y)
                            (vx, vy) in
  play (x', y') (vx, vy)
let () =
  Draw.init();
  let speed = 0.1 in
  let vx = speed *. Random.float 1. in
  let vy = speed *. Random.float 1. in
  play (Draw.right /. 2.,
        float Draw.thick)
       (vx, vy)
```

```
ocamlopt -o breakout graphics.cmxa draw.ml breakout.ml
```

ou

```
ocamlc -o breakout graphics.cma draw.ml breakout.ml
```

```
open Graphics
let left = 0.
let right = 300.
let down = 0.
let up = 200.

let ball = 5
let paddle = 50
let thick = 8

let gray = rgb 220 220 220
```

começamos por definir as dimensões do tabuleiro (as 4 primeiras constantes)

são flutuantes, por precisarmos de realizar cálculos deste tipo para as trajetórias

as constantes `ball`, `paddle` e `thick` definem a bola (raio) e a raquete (largura, comprimento) em píxeis

finalmente define-se a cor (cinzenta - na forma *rgb*)

a componente gráfica - a inicialização

```
let init () =
  let s = Printf.sprintf
            " %dx%d"
            (truncate right)
            (truncate up)  in
  open_graph s;
  auto_synchronize false
```

a função `init` inicializa a o tabuleiro abrindo uma janela gráfica de dimensão $\lfloor$`right` $\times$ `left`$\rfloor$

um detalhe particular da animação gráfica com `Graphics` obriga à utilização do utilitário `auto_synchronize` que tentaremos aqui somente sugerir e não pormenorizar:

permite melhorar o desempenho do *refresh* do ecrã a cada ronda do algoritmo (i.e. a animação) usando um modo de *buffering* especial e evitar o efeito "*pisco*"

em contrapartida, de cada vez que se pretende actualizar o desenho, é preciso usar a função `synchronize`

a componente gráfica - uma ronda

```
let clear () =
 set_color gray;
 fill_rect 0 0
   (truncate right) (truncate up)

let get_paddle_pos () =
 let x = fst (mouse_pos ()) in
 max 0 (min x
         (truncate right - paddle))
```

o passo 2 do algoritmo é realizado pela função `clear`

esta apaga a janela desenhando por cima um rectângulo cinzento do mesmo tamanho

o passo 3 do algoritmo é implementado pela função `get_paddle_pos`

esta começa por adquirir a abscissa da posição corrente do rato (via `mouse_pos`) para assegurar que a posição obtida não saia da janela de jogo majoramos o resultado obtido por `(truncate right - paddle)`

```
let game x y =
  clear ();
  set_color black;
  fill_circle (truncate x)
      (truncate y) ball;
  let x = get_paddle_pos () in
  fill_rect x 0 paddle thick;
  synchronize ();
  x
```

a função game executa de forma coordenada os passos 2, 3 e 4 do algoritmo

a componente jogo - uma ronda

```
let bounce (x, y) (vx, vy) xp =
  let vx =
    if  x <= Draw.left
     || x >= Draw.right
    then -. vx else vx in
  let vy =
    if   y <= float Draw.thick
     && x >= xp &&
      x <= xp +. float Draw.paddle
      || y >= Draw.up
    then -. vy else vy
  in
  (vx, vy)

let new_position (x, y) (vx, vy) =
   x +. vx, y +. vy
```

o código de `breakout.ml` implementa os passos 1, 5 e 6 do algoritmo

o estado do jogo está definido por dois pares (x, y) e (vx, vy) que representam (resp.) as coordenadas da bola e o seu vector de velocidade e pela posição da raquete (xp)

a função `bounce` calcula o novo vector de velocidade tendo em conta o ressalto eventual nas bermas da janela ou na raquete

a actualização de vx necessita da comparação de x com a posição das duas paredes laterais

Problema: as constantes `left` e `right` úteis neste contexto estão definidas no ficheiro `draw.ml`

a componente jogo - uma ronda

```
let bounce (x, y) (vx, vy) xp =
  let vx =
    if  x <= Draw.left
     || x >= Draw.right
    then -. vx else vx in
  let vy =
    if   y <= float Draw.thick
     && x >= xp &&
      x <= xp +. float Draw.paddle
      || y >= Draw.up
    then -. vy else vy
  in
  (vx, vy)

let new_position (x, y) (vx, vy) =
   x +. vx, y +. vy
```

cada unidade de compilação define um módulo (como `List` ou `String`) cujo nome é o do ficheiro (começando por uma maiúscula)

$\Longrightarrow$ **Draw.left** **Draw.right** **...**

para calcular `vx` tomamos conta de `left` e `right` do módulo `Draw`

da mesma forma, para calcular `vy`, tomamos conta da parede superior (`Draw.up`) e da posição da raquete (`x`, entre `xp` e `xp +. (float Draw.paddle)`, e `y` inferior a `Draw.thick`)

a função `new_position` calcula a nova posição tendo em conta a posição atual e o vector de velocidade (o par resultando da soma de `x` com `vx` e de `y` com `vy`)

a componente jogo - uma ronda

a função `play` recebe o estado corrente do jogo e implementa o ciclo do algoritmo (passos 2 a 6)

```
let rec play (x, y) (vx, vy) =
 if y <= Draw.down
   then failwith "game over";
 let xp = Draw.game x y in
 let vx', vy' = bounce (x, y)
                  (vx, vy)
                  (float xp) in
 let x', y' =  new_position
                  (x, y)
                  (vx', vy') in
 play (x', y') (vx', vy')
```

começa por verificar se a bola está dentro do tabuleiro (a coordenada `y` está por cima de `Draw.down`)

caso não esteja, o jogo termina (`failwith`)

Caso esteja, desenhamos a bola e a raquete (com recurso à função `Draw.game`) que retorna a nova posição desta última

com esta informação calculamos o novo estado do jogo (novo vector de velocidade e posição da bola)

e retomamos a ronda do algoritmo, chamando a função `play` com o estado do jogo atualizado

```
let () =
  Draw.init();
  let speed = 0.1 in
  let vx =
    speed *. Random.float 1. in
  let vy =
    speed *. Random.float 1. in
  play
    (Draw.right /. 2.,
     float Draw.thick)
    (vx, vy)
```

o programa principal do jogo consiste na inicialização do tabuleiro

na inicialização (aleatória) do vector de velocidade

e da chamada a função play (a bola encontra-se inicialmente por cima da raquete no meio do tabuleiro)

no lugar de

```
ocamlc -o breakout graphics.cma draw.ml breakout.ml
```

podemos descompor a compilação em etapas:

```
ocamlc -c draw.ml                    %cria draw.cmo e draw.cmi
```

seguido de

```
ocamlc -c breakout.ml                %cria breakout.cmo e breakout.cmi
```

e finalmente

```
ocamlc -o breakout graphics.cma draw.cmo breakout.cmo
```

as duas primeiras etapas indicam ao compilador que este só se deverá preocupar em gerar código compilado sem procurar gerar um executável (i.e. um ficheiro **objecto** `.cmo` e a sua **interface objecto** `.cmi`)

a ordem importa: o segundo ficheiro refere-se ao módulo `Draw` por isso o ficheiro de interface compilado `draw.cmi` deve previamente existir, até para permitir a tipagem

finalmente, na ultima linha, realizamos a edição das ligações (***linking***) e criamos um executável

compilação separada e *linking*

é preciso perceber que um ficheiro `.cmo` contém código e eventuais chamadas a código externo

por exemplo o código do ficheiro `draw` invoca funções de `graphics`, como o ficheiro breakout chama utilitários definidos em `draw`

na compilação propriamente dita, a tradução do código de `breakout` (por exemplo) não precisa de conhecer os detalhes (a constituição) dos utilitários de que precisa de `draw`

só tem de saber a assinatura tipada (presente nos ficheiros `cmi`) para realizar adequadamente as chamadas e confirmar a boa tipagem (...existe! e tem esta forma)

para gerar um executável, é desta vez necessário conhecer o código repartido nos vários ficheiros envolvidos e agrupa-lo adequadamente (**dizer fisicamente onde está o quê**) : é o que faz a edição das ligações

```
ocamlc -o breakout graphics.cma draw.cmo breakout.cmo
```

nota final: o executável é a sequência das expressões compiladas contidas nos ficheiros da linha de comando da compilação percebe-se porque a **ordem é relevante**

olhemos para a constituição do ficheiro `draw.cmi`

este contém **por omissão** todas as informações de tipagem que a opção -i do compilador também produz

```
> ocamlc -i draw.ml
  val left : float
  val right : float
  ...
  val game : float -> float -> int
```

podemos querer expor menos informação para os utilizadores do módulo `Draw` por exemplo é interessante não publicar a existência da função `clear` visto ser uma função de utilidade local

para tal, podemos editar um ficheiro de **interface** para `draw.ml`, o ficheiro `draw.mli`
com uma interface, controlamos o que queremos tornar **público** e **disponível** externamente

Breakout Game - draw.mli

o ficheiro `draw.mli`

```
val left : float
val right : float
val down : float
val up : float

val paddle : int
val thick : int

val init : unit -> unit
val game : float -> float -> int
```

a compilação faz-se da seguinte forma

```
> ocamlc -c draw.mli
> ocamlc -c draw.ml
```

a primeira ordem de compilação cria o ficheiro `cmi` correspondente

a segunda ordem de compilação verificará também se os tipos das expressões presentes estão de acordo com as declarações presentes no `.mli`

uma nota: se o compilador detecta a presença de um ficheiro `mli`, este recusa compilar o ficheiro `ml` correspondente sem uma compilação prévia do `mli` (i.e sem uma geração adequada dum `cmi`)

qualquer programa que utilize os utilitários de `draw` só poderá aceder aos nomes declarados no `mli`

numa primeira exposição aos módulos, refere-se que estes têm uma granulosidade mais fina do que a do ficheiro: um ficheiro `ml`, por omissão, define um módulo, mas podemos definir vários módulos num ficheiro

```
module type I = sig
   val a: int
   val f: int -> int
end
```

```
module M :  I = struct
   let a = 42
   let b = 1729
   let f  x = a * x + b
end
```

```
# M.f 6;;
- : int = 1981
# M.a;;
- : int = 42
# M.b;;
Error: Unbound value M.b
```

Tartaruga Logo

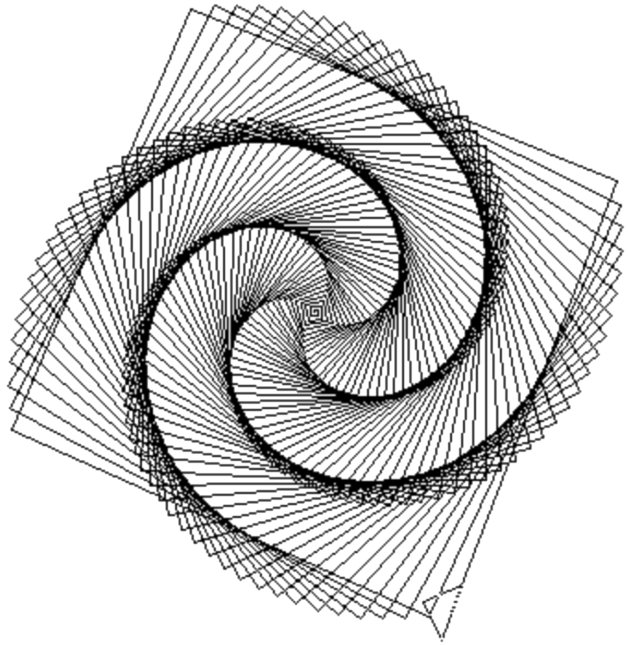
noções por introduzir neste exemplo

- tipos abstractos
- tipos privados
- encapsulamento
- functores

a **tartaruga Logo**, popular no seu tempo, foi um ambiente para a aprendizagem da programação, como hoje temos o `scratch`

o conceito: uma tartaruga num tabuleiro com um lápis

o programa dá ordens de movimentação à tartaruga (avançar x unidades, rodar de y graus, levantar o lápis, baixar o lápis, etc.) e conforme o programa, é observado o rasto deixado pelo lápis durante a movimentação da tartaruga



escrever um pequeno subconjunto simples mas representativo do ambiente de execução de programas Logo

primeiro desafio: como representar os ângulos que determinam a direção que a tartaruga tomará?

varias escolhas são possíveis (na forma de um inteiro, de um flutuante? graus ou angulo radiano? etc.)

podeos simplesmente não ecolher e escrever código OCaml parametrizado pela forma com que pretendemos lidar com os ângulos: via um módulo

```
module type ANGLE = sig
  type t
  val of_degrees : float -> t
  val add : t -> t -> t
  val cos : t -> float
  val sin : t -> float
end
```

tipo dos ângulos

neste modulo *assinatura*, o tipo `t` é o tipo dos ângulos

não tem definição: dizemos que é um tipo **abstracto**

a função `of_degree` permite usar elementos do tipo `t` a partir de um valor numérico (flutuante) que representa um ângulo em grau

se, posteriormente, escolhermos representar os ângulos como graus numa representação flutuante, então `t=float` e `of_degree` é a função identidade

```
module type ANGLE = sig
  type t
  val of_degrees : float
  val add : t -> t -> t
  val cos : t -> float
  val sin : t -> float
end
```

a solução escolhida que permite dar toda a liberdade ao programador que bem pode querer outro tipo de representação

podemos então escrever o módulo `Turtle` como sendo um módulo parametrizado pela forma como que escolhemos lidar com os ângulos

dizemos do módulo `Turtle` que é um **functor** porque pode ser visto como um operador que aceita um módulo em entrada e que devolve outro módulo adaptado ao modulo fornecido

```
module Turtle (A:ANGLE) = struct...
```

dentro do módulo `Turtle` o módulo `A` cuja assinatura é `ANGLE` está visível e utilizável

```
let angle = ref (A.of_degree 0.) ...
let rotate_left d = angle := A.add !angle (A.of_degrees d)
...
```

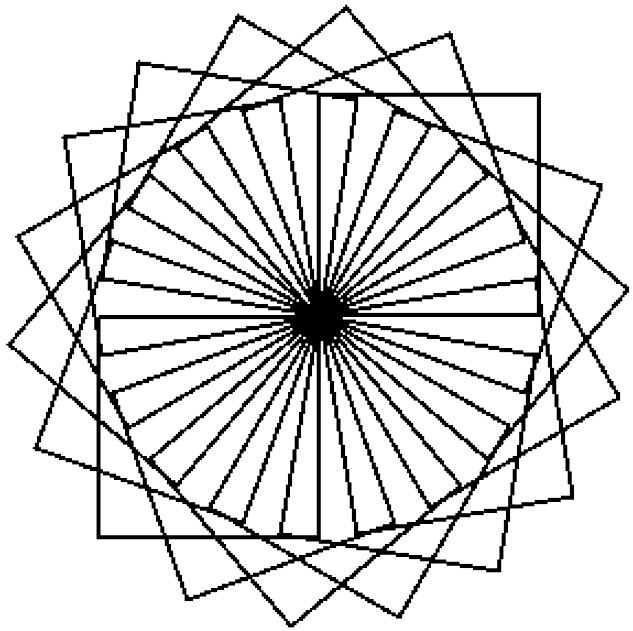
turtle_logo.ml

```
module type ANGLE = sig
  type t
  val of_degrees: float -> t
  val add: t -> t -> t
  val cos: t -> float
  val sin: t -> float
end

module Turtle(A: ANGLE) = struct

 let draw = ref true
 let pen_down () = draw := true
 let pen_up   () = draw := false

 let angle = ref (A.of_degrees 0.)


(* continua na coluna seguinte *)
```

```
 let rotate_left d =
     angle :=  A.add !angle (A.of_degrees d)
 let rotate_right d = rotate_left (-. d)

 open Graphics
 let tx = ref 400.
 let ty = ref 300.
 let () = open_graph " 800x600";
  moveto 400 300; set_line_width 2

 let advance d =
   tx := !tx +. d *. A.cos !angle;
   ty := !ty +. d *. A.sin !angle;
   if !draw
      then lineto (truncate !tx) (truncate !ty)
      else moveto (truncate !tx) (truncate !ty)

end
```

exemplo: logo.ml

```
open Turtle_logo

module Angle: ANGLE =  struct
     type t = float

     let add = (+.)
     let pi_over_180 = atan 1. /. 45.
     let of_degrees d = d *. pi_over_180
     let cos  = Pervasives.cos
     let sin = Pervasives.sin
   end

module T : Turtle_logo.Turtle(Angle)


let square d =
  for k = 1 to 4 do T.advance d; T.rotate_left 90. done

let squares d a = for k = 1 to truncate (360. /. a) do
    square d; T.rotate_left a
  done; ignore (Graphics.read_key ())


let () = squares 100. 20.
```



resultado

```
> ocamlopt graphics.cmxa -c turtle_logo.ml
> ocamlopt graphics.cmxa turtle_logo.cmx logo.ml -o logo
```

o interesse em definir o módulo `Turtle` desta forma é o de poder **encapsular** a representação dos ângulos

o módulo `Turtle` abstraí-se da forma com que escolhemos representar os ângulos desde que uma determinada **interface** definida (funções, valores **públicos**) seja respeitada

a declaração desta interface é dada pelo módulo assinatura `ANGLE`, uma representação possível foi dada pelo módulo `Angle`, e uma instanciação (uso) pelo functor `Turtle` é dada pelo módulo `T` (resultado do functor `Turtle` aplicado a `Angle`)

como em outros paradigmas (e.g. OO), é **transparente trocar a implementação dos ângulos** sem impactar de forma alguma o código que dele depende

o tipo dos ângulos, `A.t`, é um **tipo abstracto** o sentido que nos é dado a sua existência mas não o detalhe da sua real definição

pela já referida noção de encapsulamento, um tipo abstracto pode servir igualmente para esconder... abstrair-se de... uma implementação mesmo quando esta é conhecida

por exemplo, imaginemos que queiramos representar o intervalo inteiro $\{0 \ldots 30\}$

neste caso podemos introduzir esta assinatura

```
module type INT31 = sig
  type t
  val create : int -> t
  val value : t -> int
end
```

```
module Int31 : INT31 = struct
  type t = int
  let check x = if x < 0 || x > 30
                then invalid_arg "Int31.create"
  let create x = check x ; x
  let value x = x
end
```

dentro deste módulo, concretizamos o tipo abstracto `t` pelo tipo `int`

por fora, `t` é abstracto, i.e. em particular não sabemos – não podemos usar o facto – que `t` é implementado com recurso ao tipo inteiro clássico.

```
# let x = Int31.create 7;;
x : Int31.t = <abstr>
# x + 10;;
Error: This expression has type Int31.t
         but an expression was expected of type int
# Int31.value x + 10;;
- : int = 17
```

uma consequência interessante é que podemos assim garantir propriedades (designados de **invariantes**) particulares sobre os valores do tipo `Int31.t` que não poderíamos garantir no caso geral do tipo `int`

aqui, obviamente que $\forall x \in Int31.t,\ 0 \leq x \leq 31$

os functores representam um mecanismo poderoso e elegante para a programação modular e genérica

o seu uso na biblioteca padrão de OCaml é recorrente

muitas estruturas de dados fornecidas por esta biblioteca são parametrizadas via functores

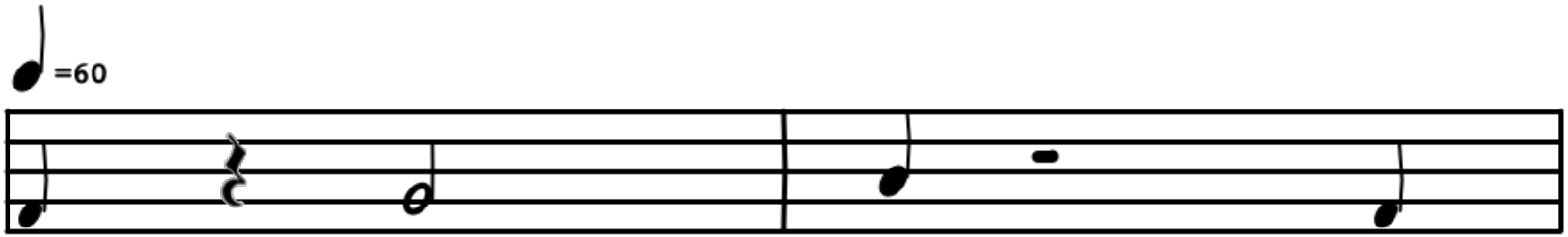
algoritmos podem também ser parametrizados pelos mesmos meios, obtendo assim implementações que se adaptam de forma muito elegante às mais diversas situações

um exemplo pode ser um functor que, via uma parametrização adequada, forneça os mecanismos gerais da técnica de *backtracking* adaptados ao caso considerado pelo programador

Tocar uma partitura musical

noções por introduzir neste exemplo

- tipos algébricos
- a construção `match` e filtragem



vamos desenhar um programa que permite tocar partituras musicais muito simples

para tal é preciso distinguir alguns conceitos linguísticos no mundo musical

- notas. Determinadas pela (altura da) posição na partitura e pela duração. Interessamo-nos aqui somente a dois tipos de duração para simplificar o exemplo
- silêncio. De forma semelhante às notas, queremos representar momentos de silêncio pela sua duração (na mesma medida)

falta-nos aqui referir o *tempo*, que é o numero de semínimas por tocar num minuto (no exemplo, é 60 por minuto)

para tocar uma nota precisamos determinar o som adequado, isto é, a sua frequência (em Hz), para tal indicamos:

- a **nota** principal (i.e. **do**, **re**, **mi**, **fa**, **sol**, **la** ou **si**)
- a oitave (0, 1, 2, 3 ..)

com base nesta informação é fácil saber a frequência exata de uma nota:

$$f = f_0 \times 2^o$$

onde f_o é a frequência na oitava 0 e o é a oitava pretendida

nota	do	re	mi	fa	dol	la	si
Freq. oitava 0 (Hz)	33	37	41	44	49	55	62

Tipo algébrico para a representação das notas

para representar as notas utilizamos um tipo de dado definido especificamente para o propósito, nomeadamente um **tipo algébrico**

os tipos algébricos de base são **enumerações finitas** dos elementos constituintes

```
type note = Do | Re | Mi | Fa | Sol | La  | Si
```

cada um dos elementos é definido por um **construtor**

OCaml obriga a que o identificador de um construtor seja único e comece por uma maíuscula

```
# Sol;;
- : note = Sol
```

a altura das notas é representada pelo tipo registo `pitch`

```
type pitch = { note : note; octave : int }
```

quanto à duração das notas, assumimos aqui só dois tipos: a mínima (meio tempo) e a semínima (quarto de tempo)

```
type  duration = Half | Quarter
```

para representar os símbolos que podem constar numa pauta definimos o tipo **algébrico** `symbol` seguinte:

```
type  symbol = Note of pitch * duration | Rest of duration
```

notemos que, à diferença dos outros exemplos, os construtores têm aqui parâmetros (introduzidos pelo tipo da informação que requeremos estar associado a cada caso)

em `C`, poderemos recorrer ao mesmo mecanismo via os tipos `union`

resta-nos definir o tipo das pautas, aqui designado de `score` Este é uma estrutura que guarda a lista das notas (`symbols`) e o *tempo* da partitura (de tipo `int`)

```
type score = { symbols : symbol list;  metronome : int }
```

```
# Half;;
- : duration = Half
# Rest Quarter;;
- : symbol = Rest Quarter
# Note ({note= Si; octave = 1}, Quarter);;
- : symbol = Note ({note = Si; octave = 1}, Quarter)
```

```
let frequency {note = n; octave = o}=
  let f0 =
    match n with
      | Do -> 33
      | Re -> 37
      | Mi -> 41
      | Fa -> 44
      | Sol -> 49
      | La -> 55
      | Si -> 62
  in
   f0 * truncate (2. ** float o)
```

utilizamos aqui a construção **match** para perceber que valor é o valor `n` (de tipo `note`) e determinar assim a frequência na oitava 0

com esta informação é fácil, tendo em conta a oitava, calcular a frequência da nota por tocar:

$$f_o \times \lfloor 2^o \rfloor$$

a segunda função calcula o tempo total associado a uma duração, tendo em conta um dado *tempo* t para a partitura por tocar

```
let millisecondes d t =
  let quarter = 60000 / t in
  match d with
    | Half -> quarter * 2
    | Quarter -> quarter
```

começamos por calcular o tempo de uma semínima em milisegundos ($quarter = \frac{60000}{t}$ — há 60.10^3 milisegundos num minuto)
se a nota for uma semínima, o tempo total é *quarter* senão, é uma mínima, e neste caso o tempo é $2 \times quarter$

voltamos a usar a comodidade fornecida pelo **match** para **destruturar** / **determinar** valores de tipo `duration`

a função `sound` seguinte toca conforme os seus parâmetros (o tempo `t` e uma nota `s`) ou uma nota musical ou impõe um silêncio de uma duração determinada

usamos novamente a comodidade oferecida pelo match para saber em que caso estamos e extrair a informação útil para prosseguir com o efeito desejado

tanto no caso de uma nota como de um silêncio, calculamos a frequencia e o tempo(frequência `f` - ou 0 no caso dum silêncio - e duração calculada com base na chamada à função `millisecond`)

requeremos então que a função `sound` (do módulo `Graphics`) toca o respectivo som

```
let sound t s =
  let f,m = match s with
              | Note (p, d) -> frequency p, (millisecondes  d t)
              | Rest r      -> 0,(millisecondes  r t)
  in Graphics.sound f m
```

tendo em conta uma partitura (de tipo `score`, que contem uma lista de notas e um *tempo*), a função `play_score` invoca repetidamente a função `sound` sobre as notas consideradas

```
let play_score {symbols = l; metronome = t} =   List.iter (sound t) l
```

music.ml

```
type note = Do | Re | Mi | Fa | Sol | La | Si
type pitch = { note : note; octave : int }
type duration = Half | Quarter
type symbol = Note of pitch * duration | Rest of duration
type score = { symbols : symbol list;  metronome : int }
```

```
let frequency {note = n; octave = o}=
  let f0 =
    match n with
      | Do -> 33
      | Re -> 37
      | Mi -> 41
      | Fa -> 44
      | Sol -> 49
      | La -> 55
      | Si -> 62
  in
   f0 * truncate (2. ** float o)
```

```
let millisecondes d t =
  let quarter = 60000 / t in
  match d with
    | Half -> quarter * 2
    | Quarter -> quarter

let sound t s =
  match s with
    | Note (p, d) ->
        let f = frequency p in
        Graphics.sound f (millisecondes d t)
    | Rest r ->
        Graphics.sound 0 (millisecondes r t)

let play_score {symbols = l; metronome = t}=
     List.iter (sound t) l
```

complementos sobre tipos estruturas, algébricos e filtragem

vamos introduzir como usar a proveito a construção `match` e algumas das suas extensões sobre tipos algébricos

imaginemos os dois tipos e a expressão seguintes

```
type t = A of int * float | B of {name : string; age : int}
type u = {a : t; b = int * t }

let value = { a=A (5,10.5) ; b=(6, B {age=42; name="the answer"})}
```

então

```
let f v =
  match v with
  | { b = (_ ,A (_,x) )} -> x
  | _ -> assert false (* para v=value, esse é o ramo escolhido *)

let f v =
  match v with
  | { b = (_ ,B {age=x} )} -> x
        (* para v = value, esse é o ramo escolhido. x = 42*)
  | _ -> assert false
```

caso pretendamos tratar da mesma forma um número particular de casos diferentes numa construção match podemos recorrer ao açúcar sintáctico seguinte

```
let evenlengthunder9 l =
  match l with
  | [] | [_;_] | [_;_;_;_;] | [_;_;_;_;_;_] | [_;_;_;_;_;_;_;_;_] -> true
  | _ -> false
```

as e when nos padrões

```
(* value = { a=A (5,10.5) ; b=(6, B {age=42; name="the answer"})}*)

let f v =
  match v with
  | {a = A (_,x) as y; b = _, B {name=_;age=_}} -> (x,y)
            (* para v = value, esse é o ramo escolhido. *)
            (* resposta :  *)
            (* - : float * t = (10.5, A (5, 10.5)) *)
  | _ -> assert false

let f v =
  match v with
  | {a = _; b = _, B {name=_;age=y}} when y < 50-> y
            (* para v = value, esse é o ramo escolhido. *)
            (* - : int = 42   *)
  | _ -> assert false

let f v =
  match v with
  | {a = _; b = _, B {name=_;age=y}} when y > 50-> y
  | _ -> assert false  (* para v = value, esse é o ramo escolhido.*)
```

extensões e exaustividade na construção `match`

lembrete: a construção `match` introduz uma expressão (condicional)

logo:

- uma expressão `match` tem **um valor** e tem **um tipo**
- o valor depende da condição (argumento) do `match`, logo da forma que este tem
- os padrões escolhidos para destruturar o argumento do `match` são, **no seu conjunto**, **exaustivos**
- em caso de sobreposição, o **primeiro padrão possível é o padrão escolhido**
- todos os ramos devolvem valores do **mesmo tipo**, o tipo do resultado global do `match`

a construção `when` introduz uma escolha para além do padrão ao qual está associada (quando é verificada e quando não é verificada)

logo é preciso não esquecer do caso quem que o padrão é possível mas a condição do `when` não é respeitada

extensões e exaustividade na construção match

```
let rec conta_par l n =
  match l with
| []                          -> n
| el::li when el mod 2 = 0 -> conta_par li (n+1)
|  _::li                      -> conta_par li n;;
```

vs.

```
let rec conta_par l n =
  match l with
| []      -> n
| el::li -> conta_par (if el mod 2 = 0 then n+1 else n);;
```

como visto em vários exemplos anteriores, podemos definir tipos que dependem de outros tipos mas de cujas dependências estas se abstraem

um exemplo possível é o tipo das listas

para calcular o comprimento de uma lista, concatenar duas listas (etc.) o conteúdo explícito não tem relevância

```
type ('a,'b,'c) dummy =
   Empty | C1 of  'a*int | C2 of  'a*'b | C3 of int*'c

let v = C3 5,''ola''
```

```
# let v = C3 (5,"ola");;
val v : ('a, 'b, string) dummy = C3 (5, "ola")
```

nota-se que a simples aplicação do construtor `C` não permite instanciar as variáveis de tipo `'a` e `'b`

```
type 'a option = None | Some of 'a
```

este tipo, presente na biblioteca standard do OCaml, permite elegantemente extender as funções parciais em funções totais: nos valores em que uma função não está definida (é parcial), é-lhe proposto que devolva o valor `None`

os valores habituais do contradomínio são os valores do tipo `'a` (mas colocados no construtor `Some`)

de notar que o construtor `None` não se confunde, **nunca!**, com um valor do tipo `'a`: $None \notin 'a$

... sempre é mais elegante do que um valor -1 para representar um valor de erro quando se calcula sobre inteiros positivos.

```
# let rec findpos n i l =
    match l with
      | []                        -> None
      | el :: li when el = n -> Some i
      | _::li                     -> findpos n (i+1) li;;
val findpos : 'a -> int -> 'a list -> int option = <fun>
# findpos 5 0 [1;2;3;4;5;6;7];;
- : int option = Some 4
# findpos 5 0 [1;2;3;4;6;7];;
- : int option = None
```

Árvores quaternárias

noções por introduzir neste exemplo

- árvores, árvores binárias e afins
- partilha

as **árvores binárias** codificam elegantemente estruturas de decisões sobre informação unidimensional (e.g. inteiros, vectores, listas etc.)

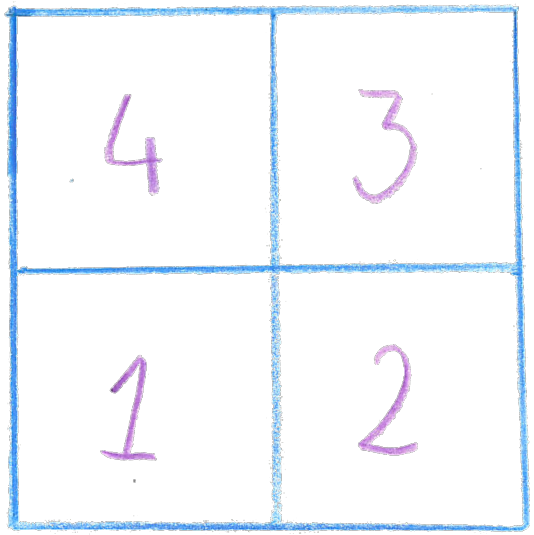
ser menor ou maior, estar a esquerda ou a direita, estar para frente ou para tràs....
estas árvores codificam a natureza binária das decisões em causa e suportam por exemplo estratégias óptimas de pesquisa/ordenação neste tipo de estruturas

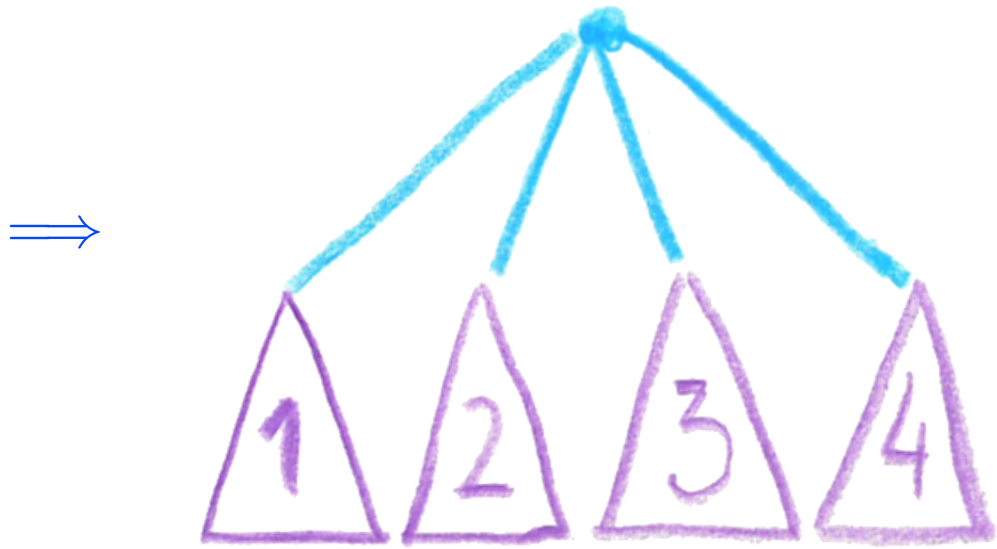
no caso de estruturas **bi-dimensionais** é natural a estrutura de decisão óptima se expressar na forma **quaternária**

como é também natural essa ser **octal** no caso de estruturas **tri-dimensionais**

no caso bi-dimensional falaremos de **árvores quaternárias** (**quad trees**) e no caso tri-dimensional de árvores octais (**octa-trees**)

divisão do espaço bi-dimensional



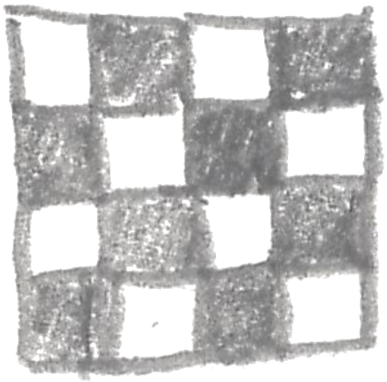


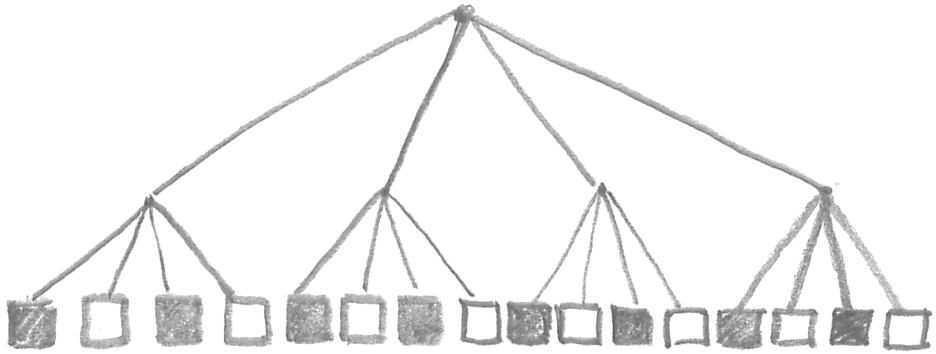
como representar eficientemente um tabuleiro?

vamos responder a esta questão escolhendo astuciosamente tabuleiros de dimensão 2^n para um dado n natural não nulo

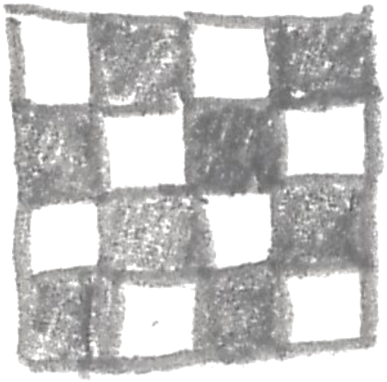
um tabuleiro, com estas dimensões, é composto por $2^n \times 2^n$ células, de cor preta ou branca.

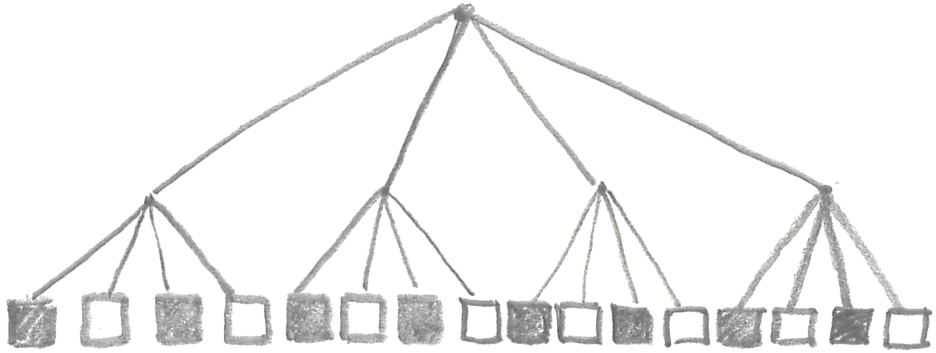
por exemplo, o tabuleiro de dimensão 4 ($2^2 \times 2^2$) seguinte pode ser representado pela árvore seguinte:





como representar eficientemente um tabuleiro?





de notar que cada *andar* da árvore representa uma potencia de 2

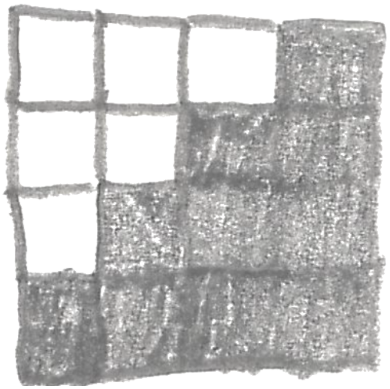
cada filho da raíz representa globalmente uma zona de 4 células do tabuleiro (na ordem estabelecida mais acima)

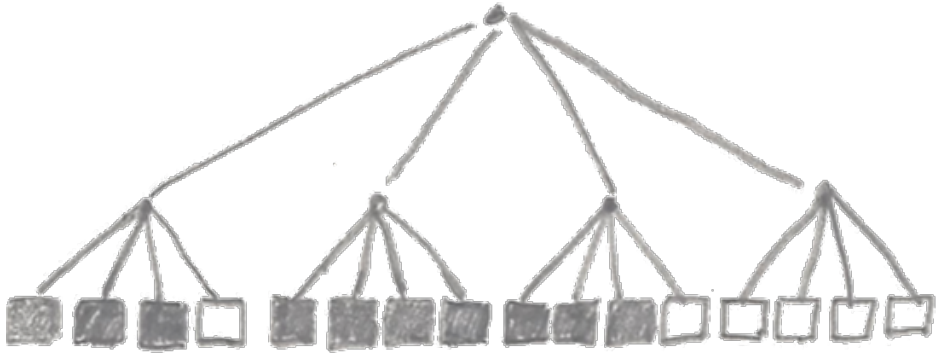
cada um deles tem 4 filhos, nomeadamente folhas, que representam as células em si

se representássemos um tabuleiro de 8 por 8 teríamos 3 andares na representação em árvore e 64 folhas

podemos melhorar drasticamente a representação de um tabuleiro ao detectar padrões e evitar com esta base representações redundantes

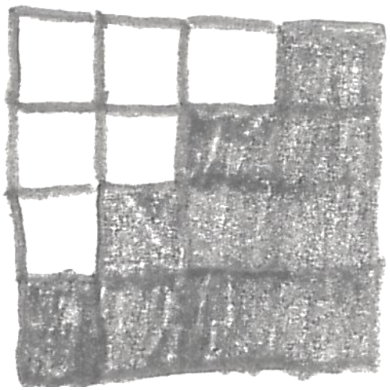
consideremos o caso seguinte





podemos melhorar drasticamente a representação de um tabuleiro ao detectar padrões e evitar com esta base representações redundantes

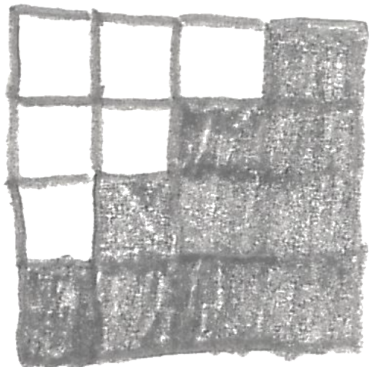
consideremos o caso seguinte

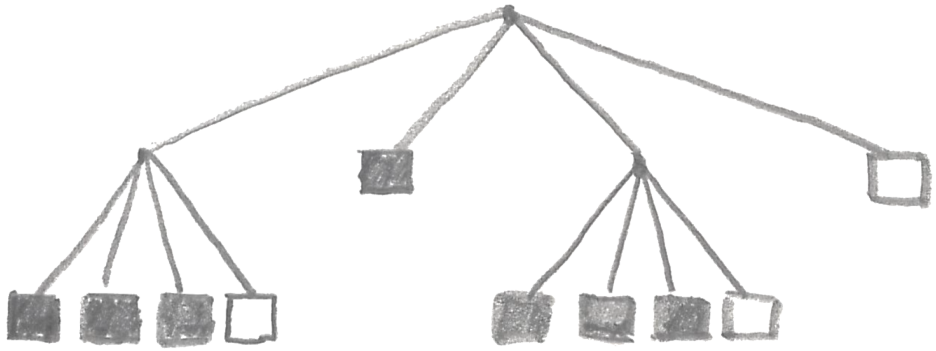


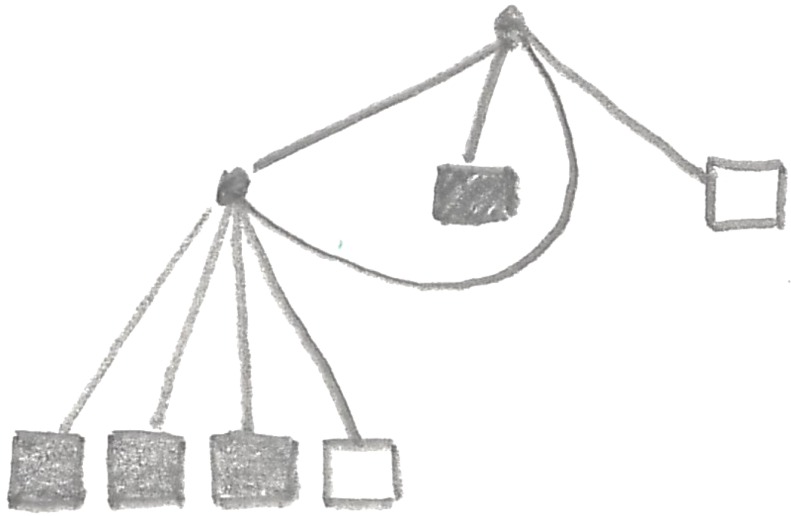
podemos decidir que um nodo cujas folhas são todos da mesma cor é logo **reduzido** a esta cor

podemos também determinar que quando dois nodos representam a mesma árvore então não são replicados (são **partilhados**)

eficiente?







podemos ir mais além ainda, utilizando a técnica da **partilha máxima** (**hash-consing**)

os curiosos poderão consultar o artigo de Sylvain Conchon e Jean-Christophe Filliâtre de 2006

Type-Safe Modular Hash-Consing

(link pdf)

Type-Safe Modular Hash-Consing

Jean-Christophe Filliâtre
LRI
Université Paris Sud 91405 Orsay France
filliatr@lri.fr

Sylvain Conchon
LRI
Université Paris Sud 91405 Orsay France
conchon@lri.fr

Abstract

Hash-consing is a technique to share values that are structurally equal. Beyond the obvious advantage of saving memory blocks, hash-consing may also be used to speed up fundamental operations and data structures by several orders of magnitude when sharing is maximal. This paper introduces an OCAML hash-consing library that encapsulates hash-consed terms in an abstract datatype, thus safely ensuring maximal sharing. This library is also parameterized by an equality that allows the user to identify terms according to an arbitrary equivalence relation.

Categories and Subject Descriptors D.2.3 [*Software engineering*]: Coding Tools and Techniques

General Terms Design, Performance

Keywords Hash-consing, sharing, data structures

1. Introduction

Hash-consing is a technique to share purely functional data that are structurally equal [8, 9]. The name *hash-consing* comes from Lisp: the only allocating function is *cons* and sharing is traditionally realized using a hash table [2]. One obvious use of hash-consing is to save memory space.

Hash-consing is part of the programming folklore but, in most programming languages, it is more a design pattern than a library. The standard way of doing hash-consing is to use a global hash table to store already allocated values and to look for an existing equal value in this table every time we want to create a new value. For instance, in the Objective Caml programming language[1] it reduces to the following four lines of code using hash tables from the OCAML standard library:

```
let table = Hashtbl.create 251
let hashcons x =
  try Hashtbl.find table x
  with Not_found → Hashtbl.add table x x; x
```

The `Hashtbl` module uses the polymorphic structural equality and a generic hash function. The initial size of the hash table is clearly context dependent and we will not discuss its choice in this paper—anyway, choosing a prime number is always a good idea.

As a running example of a datatype on which to perform hash-consing, we choose the following type `term` for λ-terms with de Bruijn indices:

```
type term =
  | Var of int
  | Lam of term
  | App of term × term
```

Instantiated on this type, the `hashcons` function has the following signature:

```
val hashcons : term → term
```

If we want to get *maximal sharing*—the property that two values are indeed shared as soon as they are structurally equal—we need to systematically apply `hashcons` each time we build a new term. Therefore it is a good idea to introduce *smart constructors* performing hash-consing:

```
let var n = hashcons (Var n)
let lam u = hashcons (Lam u)
let app (u,v) = hashcons (App (u,v))
```

By applying `var`, `lam` and `app` instead of `Var`, `Lam` and `App` directly, we ensure that all the values of type `term` are always hash-consed. Thus maximal sharing is achieved and physical equality (`==`) can be substituted for structural equality (`=`) since we now have

$$x = y \iff x == y.$$

In particular, the equality used in the hash-consing itself can now be improved by using physical equality on sub-terms, since they are already hash-consed by assumption. To do such a bootstrapping, we need custom hash tables based on this new equality. Fortunately, the OCAML standard library provides generic hash tables parameterized by arbitrary equality and hash function. To get custom hash tables, we simply need to define a module that packs together the type `term`, an equality and a hash function

```
module Term = struct
  type t = term
  let equal x y = match x,y with
    | Var n, Var m → n == m
    | Lam u, Lam v → u == v
    | App (u1,u2), App (v1,v2) →
        u1 == v1 && u2 == v2
    | _ → false
  let hash = Hashtbl.hash
end
```

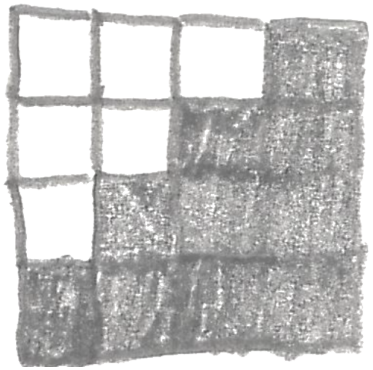
and then to apply the `Hashtbl.Make` functor:

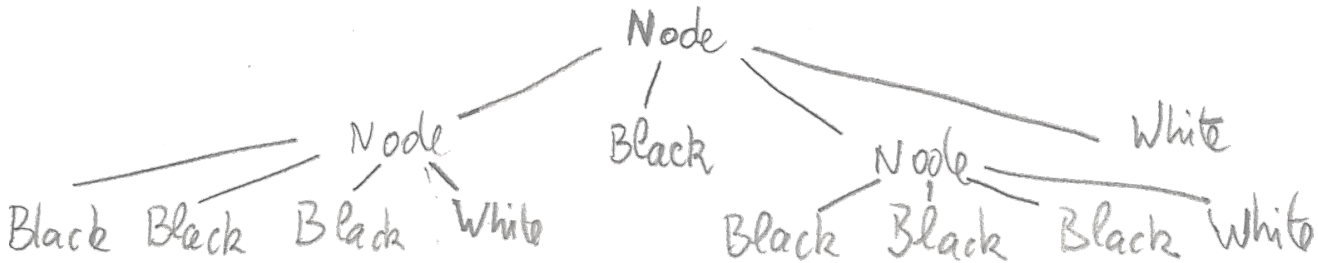
```
module H = Hashtbl.Make(Term)
```

[1] We use the syntax of Objective Caml (OCAML for short) [1] throughout this paper, but this could be easily translated to any ML implementation.

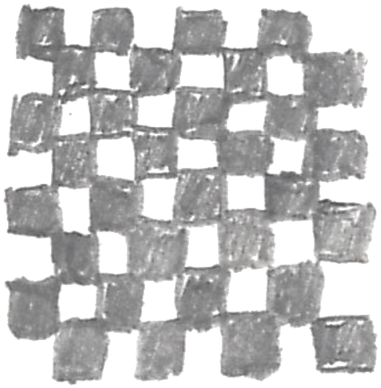
ML'06 September 16, 2006, Portland, Oregon, USA.

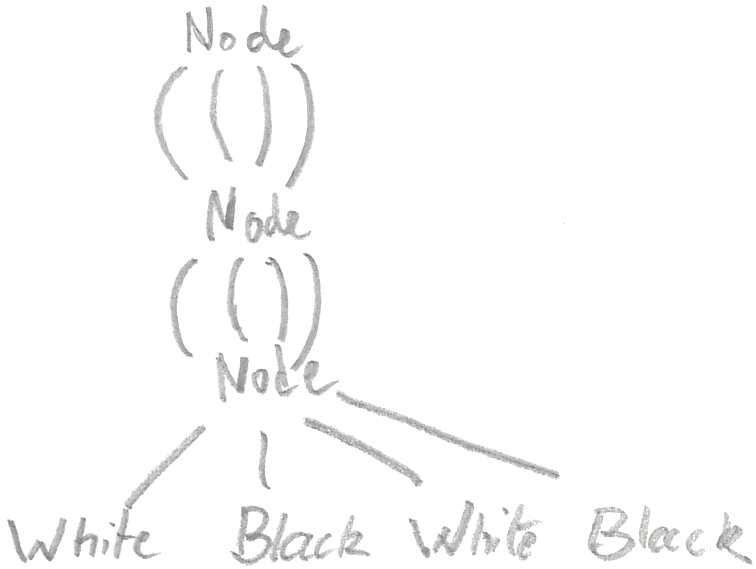
```
type quad = White | Black | Node of quad * quad * quad * quad
```





ou melhor ainda





o objectivo do código seguinte é gerar tabuleiros de tamanho $2^n \times 2^n$ para um dado n natural (função `checker_board`)

e consequentemente, vizualizar o referido tabuleiro (função `draw`).

```
type quad = White | Black | Node of quad * quad * quad * quad

let rec checker_board = function
  | 0 -> Black
  | 1 -> Node (White, Black, White, Black)
  | n -> let q = checker_board (n - 1) in Node (q, q, q, q)

let rec draw x y w = function
  | White ->  ()
  | Black ->   Graphics.fill_rect x y w w
  | Node (q1, q2, q3, q4) ->
      let w = w / 2 in
      draw  x       y       w q1;
      draw (x + w)  y       w q2;
      draw (x + w) (y + w)  w q3;
      draw  x      (y + w)  w q4

let () = let n = read_int () in
         let w = int_of_float (2. ** (2. ** (float_of_int n))) in
         let d = string_of_int w in
         Graphics.open_graph (" "^d^"x"^d);
         draw 0 0 w (checker_board n);
         ignore (read_key ())
```

se $n = 0$ escolhemos devolvemos (é uma escolha..) uma célula negra

se $n = 1$, devolvemos um tabuleiro 2×2 de base

se $n > 1$ utilizamos 4 vezes o **mesmo** tabuleiro de tamanho $2^{n-1} \times 2^{n-1}$
$\Rightarrow$ é onde **explicitamos a partilha**!
(i.e. `Node (q,q,q,q)` onde `q` é o tabuleiro de dimensão imediatamente menor)

$\Rightarrow$ já não falamos de árvore, mas sim de um gafo acíclico (DAG)

`checker_board` executa-se em **tempo e em memória em** $\mathcal{O}(n)$!!

a chamada `(draw x y w c)` visa desenhar um tabuleiro (quadrado) de dimensão w e cujo canto inferior esquerdo está posicionado em (x, y)

esta procegue por recursividade sobre a estrutura do tabuleiro por desenhar (em partyicular sobre os 4 sub-tabuleiros)

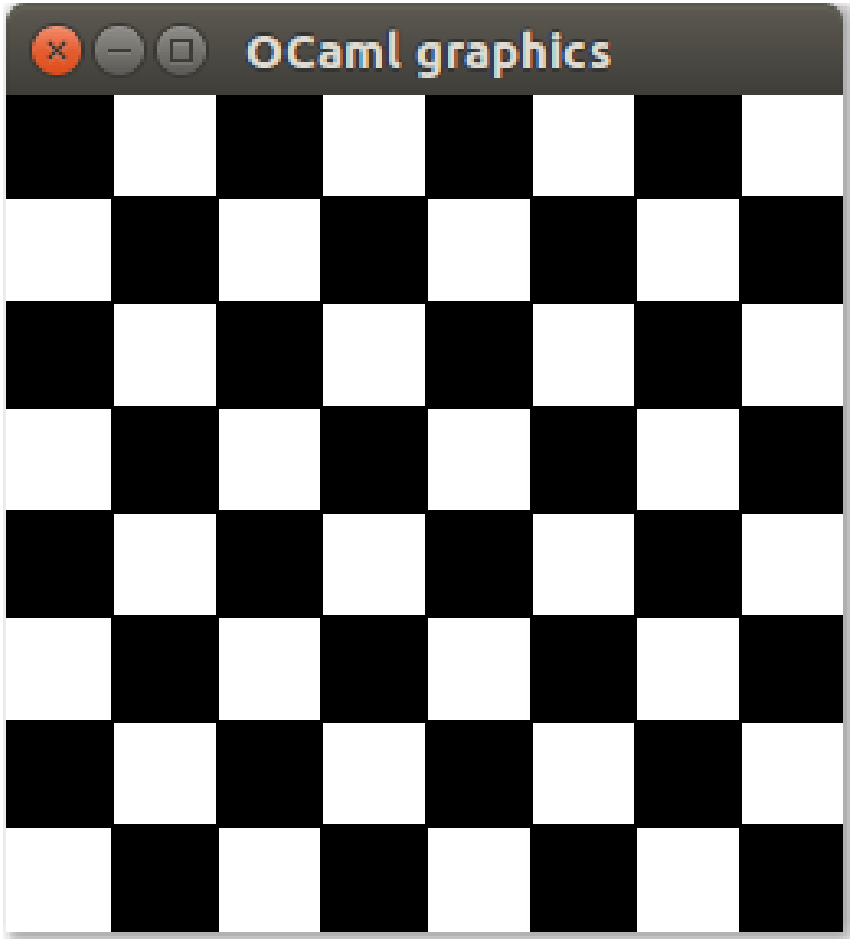
na abordagem simples aqui seguida, a função `draw` **desconhece a eventual natureza partilhada** da árvore que vai desenhar

consegue, no melhor, ter em conta o facto de um pedaço de tamanho $2^i \times 2^i$ (para um dado $i \leq n$) ser de uma só cor.

assim, a função `draw` ignora a eventual natureza *DAG* do seu parâmetro e trata-o como uma árvore quaternária, sub-arvore por sub-arvore

ou seja, **tempo de execução** $\mathcal{O}(4^n)$!!
a técnica de **hashconsing** permite uam resolução eficaz desta situação

resultado da execução para $n = 3$



(chamada a `(draw 0 0 256 c)` em que `c` é o tabuleiro)

como vimos anteriormente, construir uma árvore da forma `Node (Black, Black, Black, Black)` ou `Node (White, White, White, White)` é um desperdício

podemos simplesmente considerar no lugar a folha `Black` (ou `White`, resp.)

de uma forma geral, podemos querer impôr um invariante sobre a estrutura de dados
(isto é, querer obrigar que uma determinada propriedade seja sempre verificada qualquer que seja o elemento da estrutura de dados considerada - é a definição do que é um invariante sobre uma estrutura de dados)

classicamente a definição e a manutenção de um invariante pode ser realizada pelo uso de uma função cuja função é construir os dados conforme o invariante

genericamente estas funções designam-se de **construtor inteligente** (***smart constructor***) e a sua utilização visa substituir uma construção directa dos nodos via o construtor `Node`

```
let node = function
  | White, White, White, White -> White
  | Black, Black, Black, Black -> Black
  | q1,q2,q3,q4                -> Node (q1,q2,q3,q4)
```

complemento: árvores binárias e n-árias

árvores binárias com folhas sem informação

```
type 'a tree = Leaf | Node of 'a tree * 'a * 'a tree
```

árvores binárias com folhas informativas e nodos internos sem informação

```
type 'a tree =  Leaf of 'a | Node of 'a tree * 'a tree
```

árvore n-árias de inteiros

```
type inttree = Node of  int * inttree list
```

(neste caso as folhas são os nodos sem filhos, i.e. com lista vazia)
assumindo a existência de uma função compare sobre o tipo dos elementos a inserção ordenada sem repetição em árvores binárias com folhas sem informação, pode escrever-se

```
let rec insert_ord v = function
| Leaf -> Node (Leaf,v,Leaf)
| Node (e,el,d) as ab-> let c = compare v el in
   if c<0
   then Node ((insert_ord v e),el,d)
   else if c>0 then Node (e,el,(insert_ord v d))
   else ab
```

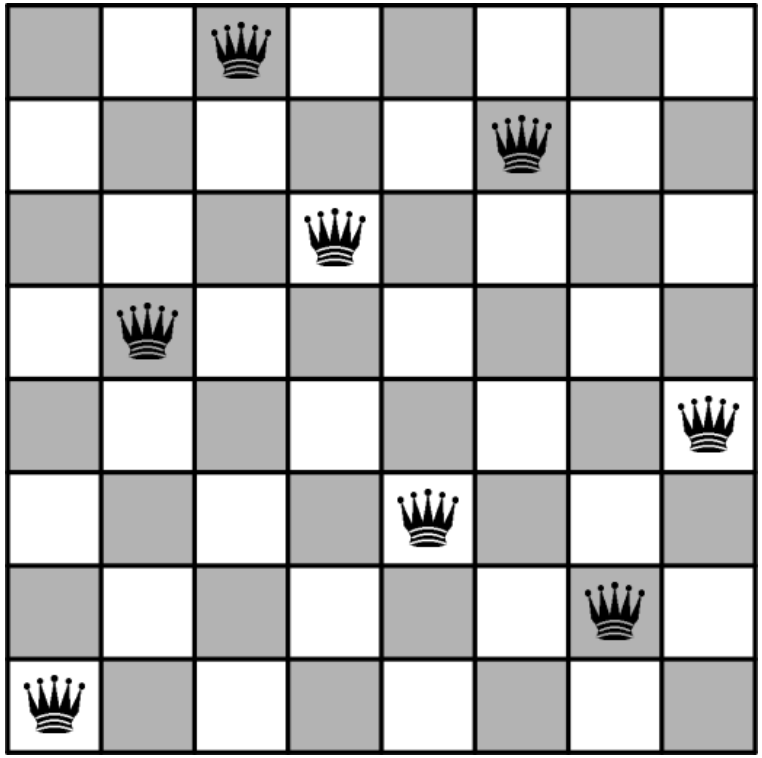
O problema das oito rainhas

```
module S = Set.Make(struct type t = int let compare = compare end)

let map f s = S.fold (fun x s -> S.add (f x) s) s S.empty

let rec upto n = if n < 0 then S.empty else S.add n (upto (n-1))

let rec count cols d1 d2 =
  if S.is_empty cols then
    1
  else
    S.fold
      (fun c res ->
        let d1 = map succ (S.add c d1) in
        let d2 = map pred (S.add c d2) in
        res + count (S.remove c cols) d1 d2)
      (S.diff (S.diff cols d1) d2)
      0

let () =
  let n = int_of_string Sys.argv.(1) in
  Format.printf "%d@." (count (upto (n - 1)) S.empty S.empty)
```

este é um problema algorítmico clássico, ao mesmo título do que as torres de Hanoí, etc.

trata-se aqui de colocar N rainhas num tabuleiro de xadrez $N \times N$ por forma a que nenhuma rainha esteja ameaçada pelas outras

problema: quantas soluções diferentes há? (i.e. excluindo soluções simétricas)



uma das 92 soluções para $N = 8$

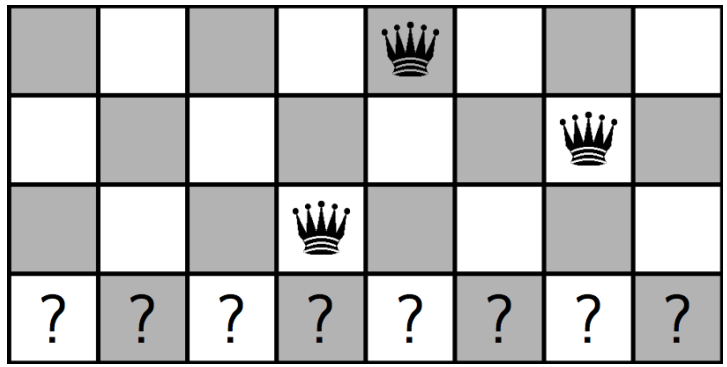
vamos tentar por força bruta auxiliada com alguma **inteligência**

há exactamente **uma e uma só** rainha **por linha**

por isso calculamos uma solução, **linha a linha**

se assumirmos que já tratamos k primeiras linhas (k rainhas que não se ameaçam mutualmente nas k primeiras linhas)

tentamos então colocar mais uma rainha nestas condições na linha $k+1$ numa posição válida



se já tratamos das 3 primeiras linhas, onde colocar a quarta rainha?

mas pode acontecer que após tratar de k rainhas (i.e. k linhas)

não seja possível colocar uma rainha em nenhuma posição da linha $k+1$ sem que esta seja ameaçada pelas rainhas já colocadas

estamos numa **beco sem saída**: com as rainhas já colocadas não temos solução

neste caso temos de **desfazer** uma escolha anterior, começando na linha anterior k (e escolher uma outra posição possível para a rainha k que não tenha sido ainda testada)

se **nem para k funciona**, então temos de desfazer a escolha na linha $k-1$, se nem aí, $k-2$ etc. se der, **recomeçar** o processo daí (etc.)

procede-se desta mesma forma até chegar a última rainha (N-ésima linha)

neste caso temos **uma solução** ao problema das N rainhas

este método que consiste um construir uma solução exploratória passo a passo em que cada passo assenta numa escolha que se pode desfazer quando há manifestação de erro é designado de

algoritmo por retrocesso - backtracking

outros exemplos: o fio de Ariadne para os labirintos, o sodoku, etc.

sabemos encontrar uma solução, como encontrar **todas** as soluções?

$\Longrightarrow$ gerar de forma **ordenada** as soluções apoiando-se na técnica de backtracking, **da primeira linha até a última linha**
e em cada linha escolher posições válidas de **uma ponta a outra** da linha

assim, inicialmente, trata-se de colocar a primeira rainha na primeira linha, numa ponta da linha
e passar à linha (rainha) seguinte etc.

quando é encontrada uma solução, esta é contabilizada

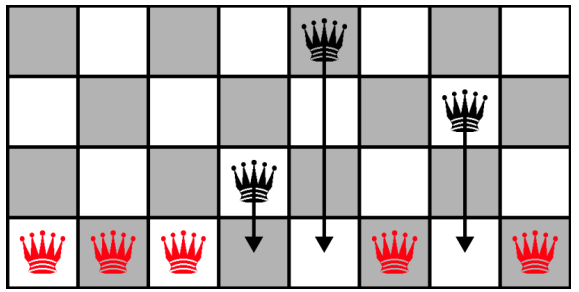
e, por *backtracking*, desfazemos ordenadamente as decisões que levaram a esta solução para encontrar a seguinte

(desfazemos a última decisão e tomamos a decisão imediatamente a seguir, se não conduzir a uma nova solução, tomamos a seguinte etc. se nenhuma decisão neste ponto for suficiente, desfazemos a antepenúltima decisão etc..)

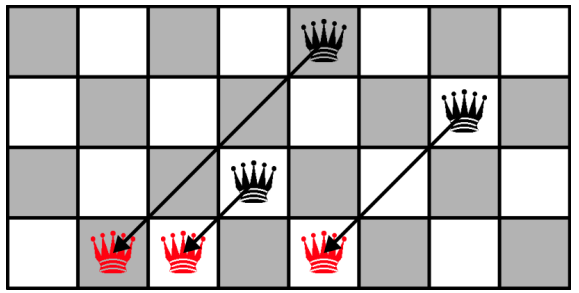
vamos na nossa implementação tentar evitar para cada linha experimentar as N colunas

como? voltemos ao caso anterior

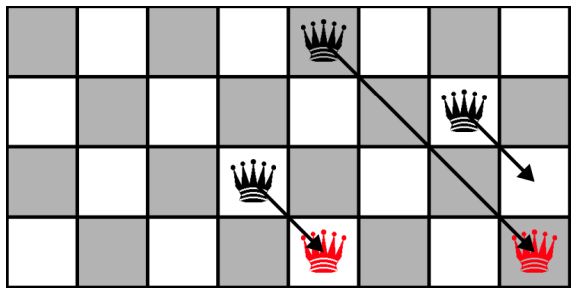
se já colocamos estas três rainhas, então podemos desde já eliminar posições possíveis (3) restando as cinco aqui destacadas



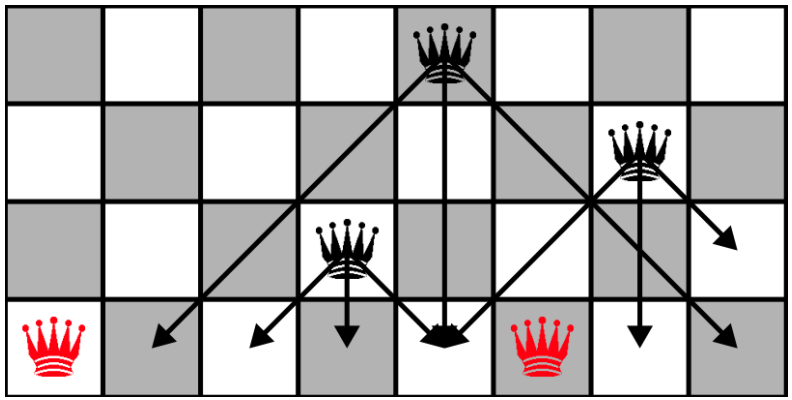
da mesma forma, podemos eliminar mais três posições devido as diagonais ascendentes



e mais essas, por causa das diagonais descendentes



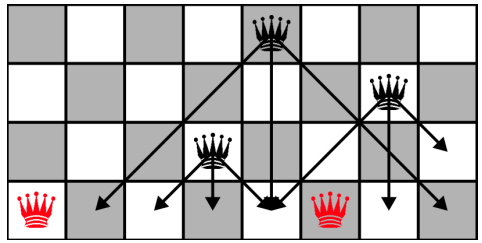
no fim resta esta configuração: só duas escolhas possíveis nesta linha nos pode levar a uma solução com as 3 rainhas anteriores



vamos a cada instante do algoritmo considerar três conjuntos

- o conjunto *cols*, das colunas ainda disponíveis
- o conjunto $d1$, das colunas indisponíveis por causa das diagonais ascendentes
- o conjunto $d2$, das colunas indisponíveis por causa das diagonais descendentes

se escolhermos numerar as colunas da direita para a esquerda começando por 0 temos no exemplo



$$cols = \{0, 2, 5, 6, 7\},\ d1 = \{3, 5, 6\}\ e\ d2 = \{0, 3\}$$

e as posições possíveis calculam-se facilmente com base na diferença de conjuntos: $cols - d1 - d2 = \{2, 7\}$

para poder trabalhar com estes conjuntos, instanciamos o módulo dos conjuntos (*aplicamos um functor*...) com o tipo dos inteiros para obter conjuntos de inteiros ordenados naturalmente

```
module S = Set.Make(struct type t = int let compare = compare end)
```

`S.t` é assim o tipo dos conjuntos de inteiros, `S.empty` é o conjunto (de inteiros) vazio, `S.diff` é a diferença sobre conjuntos de inteiros, etc.

é relevante notar que esta estrutura de dados é **persistente**:

uma operação sobre um conjunto não altera o conjunto, mas sim **devolve**, se assim for necessário, um **novo conjunto** resultante

por exemplo, imaginamos que tenhamos um conjunto de inteiro `s`,
então `S.add 4 s` devolve um **novo conjunto** que tem por elementos $s \cup \{4\}$,
em contrapartida `s` **permanece inalterado**

o nosso programa toma a forma de uma função recursiva `count` que aceita os três referidos conjuntos como parâmetros

```
let rec count cols d1 d2 =
```

e que devolve o número de soluções possíveis com base nestes três conjuntos

presenciamos uma solução quando `cols` está vazia

```
    if S.is_empty cols then
```

no caso contrário, as colunas por considerar são `(S.diff (S.diff cols d1) d2)`

```
    S.fold
      (fun c res -> ... )
      (S.diff (S.diff cols d1) d2)
      0
```

para cada uma destas colunas contamos as soluções possíveis que decorrem com uma rainha ali colocada

é o que faz `S.fold`: para todos os *c*, elementos do conjunto das colunas possíveis (`S.diff`), sabendo que até agora há *res* soluções, contar...

```
    S.fold
      (fun c res -> ... )
      (S.diff (S.diff cols d1) d2)
      0
```

mas contar como?

para cada coluna `c`, basta uma chamada recursiva à função `count` com os três conjuntos actualizados:

- para `cols`, basta remover `c` com `S.remove`
- para `d1` e `d2`, juntamos a coluna `c` (com `S.add`) e deslocar cada elementos destes conjuntos de forma adequada de uma posição (para a esquerda ou para a direita, conforma a diagonal em causa)

para a deslocação, auxiliámo-nos de uma função `map` que aplica uma função f a todos os elementos de um conjunto s ou seja $\{ f(x) \mid x \in s \}$

```
let map f s = S.fold (fun x s -> S.add (f x) s) s S.empty
```

```
      (fun c res ->
        let d1 = map succ (S.add c d1) in
        let d2 = map pred (S.add c d2) in
        res + count (S.remove c cols) d1 d2)
```

para o *shift*, deslocar-se para a direita ou para a esquerda de uma posição em colunas representadas por inteiros corresponde as operações `succ` (mais um) e `pred` (menos um)

de notar a relevância da **persistência** da estrutura de dados que suporta a representação dos conjuntos: a cada operação sobre `cols`, `d1` e `d2`, estes **não são alterados** (por `S.add` ou `S.remove`) e podem assim serem **reutilizados** sem mais cautela

é essa persistência que permite codificar **naturalmente** o **retrocesso** nas escolhas das colunas

para resolver o problema das N rainhas, basta chamar a função `count` com o conjunto $\{0, 1, 2, \ldots, N-1\}$

podemos criar tal conjunto com recurso à função `upto`

```
(ª {0,1, .. n} ª)
let rec upto n = if n < 0 then S.empty else S.add n (upto (n-1))
```

o programa principal recupera o valor de n da linha de comando e invoca `count` e `printf` de acordo

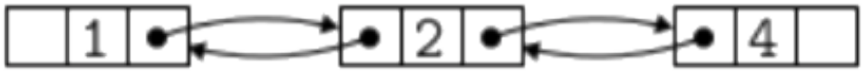
```
let () =
  let n = int_of_string Sys.argv.(1) in
  Format.printf "%d@." (count (upto (n - 1)) S.empty S.empty)
```

num minuto, é possível saber quantas soluções há para o problema das 14 rainhas

```
>  ./nqueen 14
365596
```

a literatura em estruturas de dados e algoritmos introduz na maioria estruturas ditas **imperativas**: as modificações são realizadas na **própria** estrutura (em inglês: **in-place**)

por exemplo, juntar o elemento 3 na lista duplamente ligada ordenada seguinte



dá a lista seguinte (as modificações são destacadas pelas cores)

se for necessário voltar a considerar a lista original temos de **remover/desfazer** as alterações realizadas

pode ser trabalhoso

uma estrutura de dados persistente é uma estrutura de dados cujas operações disponíveis nunca alteram a estrutura em parâmetro

as operações devolvem uma nova estrutura que contém o resultado da operação realizada

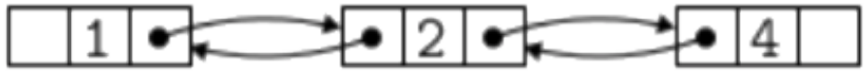
quando feito com astúcia, estas operações tem um custo reduzido: **não são necessários clones das estruturas originais**

assumindo a existência de tais estruturas, muitos algoritmos podem retirar **grande proveito** da persistência: o backtracking é um exemplo

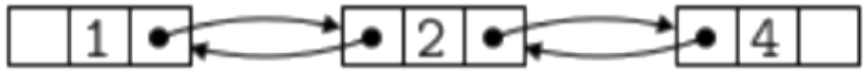
uma classe imediata de estruturas persistentes são as **estruturas imutáveis** que **não permitam** de todo **qualquer alteração**

exemplifiquemos com as listas OCaml (que são *imutáveis*)

```
let l = [1;2;3]
```




```
let l' = 0::l
```



efeito: criar **um bloco novo**, mas **nada foi copiado!**
reaproveitou-se `l` para `l'`

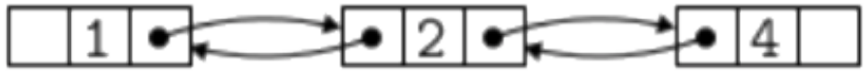
de forma geral, OCaml nunca copia estruturas imutáveis, **reaproveita**

para ter cópia, o programador tem de **explicitamente** fornecer (ou usar) **funções de cópia/clonagem**

na presença de imutabilidade, juntar na cauda é um problema

porque obriga a uma alteração da lista original


```
let rec append l1 l2 =
 match l1 with
   | [] -> l2
   | x :: l -> x :: append l l2

let l   = [1;2;3]
let l'  = [4;5]
let l'' = append l l'
```



backtracking: persistência vs. mutável

persistente

```
let rec find e =
  is_exit e || try_move e (possible_moves e)
and try_move e = function
  | [] -> false
  | d :: r -> find (move d e) || try_move e r
```

mutável

```
let rec find () =
  is_exit e || try_move e (possible_moves ())
and try_moves e = function
  | [] -> false
  | d :: r -> (move d; find ()) || (undo_move d ; try_move  r)
```

interface: persistência vs. mutável

```
type set
val empty : set
val add : int -> set -> set
val remove : int -> set -> set
...
```

```
type set
val create : unit -> set
val add : int -> set -> unit
val remove : int -> set -> unit
...
```

Conclusão. Quer saber mais?

As aulas de introdução à programação OCaml apresentadas nesta UC baseam-se em duas fontes essenciais:

- **Apprendre à Programmer avec OCaml**

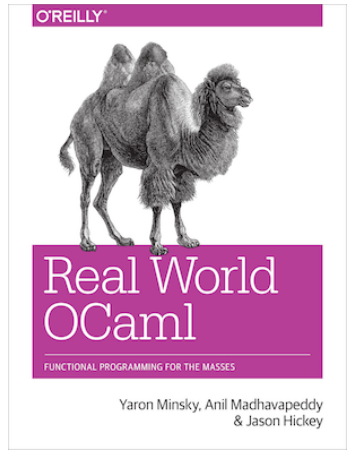
  a **fonte** deste curso!

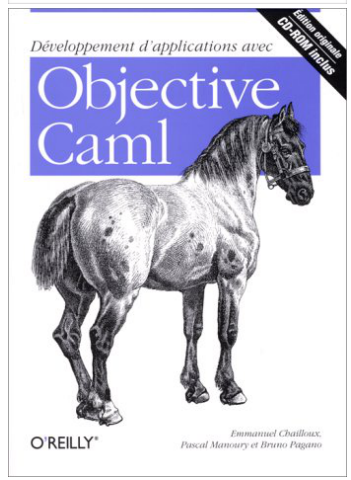
  Tradução para português disponível!

- Mini-curso **Introdução à Programação Funcional em OCaml** Simão Melo de Sousa (link)

Leitura aconselhada

Adicionalmente ou alternativamente, as referências seguintes introduzem OCaml de forma completa:





- **Real World OCaml**
- curso online: Introduction to Functional Programming in OCaml (link)

  (a aula de introdução é um espelho da aula 0 deste curso)
- **Developing Applications with Objective Caml** (pdf/html online aqui)